Cryptocurrencies (WS2024/25) Dr. Zeta Avarikioti, Prof. Matteo Maffei

Kerma - Task 4

1 General Notes

In your group of up to three students, you may use any programming language of your choice to implement the following task. We provide skeleton projects in Python and TypeScript (see `python-skeleton-for-task-4.tgz` and `typescript-skeleton-for-task-4.tgz` on TUWEL). You can use them to build upon. After the deadline, these skeleton projects will be updated and act as sample solutions for the previous tasks. These are guaranteed to pass all test cases for the previous tasks.

2 High level Overview

After completing this task, your node should be able to:

- validate any object that references unknown objects by implementing recursive object fetching.
- implement the longest chain rule.

This means that your node should implement the complete protocol with the exception of maintaining a mempool.

Intuition behind recursive object fetching

Assume your node connects to the network for the first time (thus not knowing any objects except the genesis block) and there are already many transactions and blocks in the blockchain. Using *getobject* messages, your node can request objects. But how does your node know which objects to request? How does downloading and verifying the blockchain state work?

Objects reference each other: Transactions reference other transactions (output spending), blocks reference transactions (transaction confirmation) and blocks reference their parent block. The only objects that do not rely on other objects are coinbase transactions and the genesis block, all of which are trivial to verify (coinbase transactions are valid iff they are syntactically valid, and the genesis block is valid by definition). You can only verify an object if you know that all of its referenced objects are valid. Determining dependent objects is easy - just follow the references in the objects themselves. And this is precisely what you will be doing: The protocol specifies a message *getchaintip* which lets you request the id of the topmost block from a peer (i.e., the topmost block in their view of the blockchain). You can start downloading the whole blockchain state by requesting this block, and then proceed by recursively fetching all dependent objects until you reach coinbase objects, the genesis block or valid objects that you already know about.
So you actually need to download the whole chain of blocks and all transactions referenced by them before you can verify a single object in the chain, thus leaving the validation of these objects pending after receiving them. Upon receiving the whole chain, validation proceeds in the opposite direction than you fetched it.
Should your node detect an error in any object, then all objects that (transitively) depend on this object needs to be regarded invalid (this would correspond to the error type INVALID_ANCESTRY).
Should you have not received any required object in time, then you also need to discard all (transitively) dependent objects for now (this would correspond to the error type UNFINDABLE_OBJECT). In this task, your node will need to be able to request and validate objects sent to your node in any order. See the section "Sample Test Cases" for further information.

3 Recursive Block/Chain Validation

In this task, you will build upon your object validation logic from the previous homework to be able to validate any object. When you receive an object o, perform the following steps:

- Check if you have this object stored already. If you have, than you can ignore this message.
- Perform all syntactic checks and all checks that do not require knowledge about referenced objects. If any fail, send an error message and reject it.

- Query your database for all referenced objects. Perform as many checks you can do based on the information you have on the referenced objects. Some examples:
  - If o is a block, and you know its parent block o', check if the height of o is the height of o' plus 1.
  - If o is a block, and you know all referenced transactions (and their dependencies), you can already validate the output value of the coinbase transaction (if there is one), even if you don't know the parent block of o.
  - If o is a transaction with inputs i_1, i_2, i_3 and you only know the transaction that is referenced by i_2, you can already perform the signature check for i_2.

  Again, if any checks fail send an error message and reject it.
- Determine the set of unknown objects that this object references. If there are none, validation is finished at this point (and you should broadcast knowledge of this new object to all connected peers).
- Send a *getobject* message to all of your peers for every unknown object. You should send all these messages immediately (and not fetch one object at a time).
- Start a timeout of 5s.
- Upon **receiving** * **all** unknown objects before the timeout triggers, clear the timeout.
- Upon **successful validation** of **all** unknown objects, immediately resume validation of object o. The easiest way to do this is to just restart validation of o.

Furthermore, you need to take care of the following:

- When the timeout triggers (i.e., you have not received an unknown object in time), you must release the verification of all transitively dependent objects by sending an error message UNFINDABLE_OBJECT.
- When an object turns out to be invalid, you must release the verification of all transitively dependent objects by sending an error message INVALID_ANCESTRY.
- It is fine if your node closes the connection to a peer immediately after receiving the first invalid object, and it does not have to wait until all UNFINDALBE_OBJECT or INVALID_ANCESTRY error messages have been sent.

4 Longest Chain Rule

In this exercise, you will implement the longest chain rule, and sync the chain tip with your peers.

*Why is it important to distinguish between receiving and successful validating? It is reasonable to assume that you receive objects within 5s, but it is not reasonable to assume that you can fetch and validate all dependant objects within 5s - but you have to do that before you can resume validation of this object.

- Implement the longest chain rule, i.e. keep track of the longest chain of valid blocks that you have.
- Upon connecting to a peer, send a *getchaintip* message directly after the *getpeers* message you are supposed to send.
- On receiving a `getchaintip` message from a peer, respond with a `chaintip` message with the blockid of the tip of your longest chain.
- On receiving a `chaintip` message which references an object that you do not know, request it from all of your peers (and therefore trigger validation of the whole chain). Should you know this object already and it is not a block, send an INVALID_FORMAT error and disconnect. Should the objectid already show that the object is invalid because of invalid PoW, send a INVALID_BLOCK_POW error and disconnect.
- Upon successful validation of a new block that was sent to you in an `object` message, update your longest chain if required. However, you should never remove valid objects from your database, even if they no longer belong to the longest chain.

5 Sample Test Cases

IMPORTANT: Make sure that your node is running at all times! Therefore, make sure that there are no bugs that crash your node. If our automatic grading script can not connect to your node, you will not receive any credit. Taking enough time to test your node will help you ensure this.

A note about testing

It might be useful to split testing into two parts: Test your PoW checks independently from all other checks. This allows you to then turn off PoW checking altogether during testing, so you do not have to spend computation time on building your testcases.

Below is a (non-exhaustive) list of test cases that your node will be required to pass. We will also use these test cases to grade your submission. Consider two nodes Grader 1 and Grader 2:

1. Grader 1 sends one of the following invalid blockchains by advertising a new block with the `object` message. Grader 1 must receive the corresponding error message, and Grader 2 must not receive an `ihaveobject` message with the corresponding blockid. If Grader 2 requests this object, it must receive an error message UNKNOWN_OBJECT and must not receive the object.
   (a) A blockchain that points to an unavailable block.
   (b) A blockchain with non-increasing timestamps.
   (c) A blockchain with a block in the year 2077.
   (d) A blockchain with an invalid proof-of-work in one of the blocks.

(e) A blokchain that does not go back to the real genesis but stops at a different genesis (with valid PoW but a null previd).

(f) A blockchain with an incorrect height in the coinbase transaction in one of the blocks.

(g) A blockchain with a doubling spending transaction.

(h) A blockchain with a transaction that spends an output that does not exist.

2. Grader 1 sends a number of valid blockchains. When Grader 1 subsequently sends a `getchaintip` message, it must receive a `chaintip` message with the blockid of the tip of the longest chain.

Additionally, your node will be subject to a "real-world" scenario: Your node will be started in an offline, controlled network environment, in which multiple reachable instances of the bootstrap node are running. All of them have a different longest chain, some of which are invalid. Initially, one of these nodes will connect to your node and send you a *peers* message containing all addresses of these peers. Your node is supposed to connect to each of them, request their chaintip (as specified in Section 4), validate and download their chain (as specified in Section 3) and updating your chaintip to the longest valid chain.
After your node stops sending new messages (i.e., it downloaded all of the chains) or after a reasonable timeout triggers, your node will be asked

- their chaintip, which should be the chaintip of the longest valid chain.
- their peers, which should now only consist of the peers that are valid.
- about all blocks that are used as chaintips of the peers, and your node should either
  - return this object if it is valid.
  - send an UNKNOWN_OBJECT error in case your node detected that this (or an ancestor of it) was invalid.

How to Submit your Solutions

We will grade your solution locally. Please upload your submission as a tar archive. When grading your solution, we will perform the following steps:

```
tar -xf <your submission file> -C <grading directory>
cd <grading directory>
docker-compose build
docker-compose up -d
# grader connects to localhost port 18018 and runs test cases
docker-compose down -v
```

When started in this way, there should be neither blocks (except the genesis block) nor transactions in the node's storage.

If you use the skeleton templates, you can use the provided Makefile targets to build and check your solution. Note however, that this will only check if the container can be built, started and a connection can be established to localhost:18018. We highly recommend that you write your own test cases.

The deadline for this task is 5th December, 11.59pm. We will not accept any submissions after that. Each group member has to submit the solution individually. Plagiarism is unacceptable. If you are caught handing in someone else's code, you will receive zero points.